FlexNN: Efficient and Adaptive DNN Inference on Memory-Constrained Edge Devices

背景

深度神经网络（DNN）在多个实际应用中越来越普及，如驾驶辅助、交通监控和人脸识别等。

尽管在云端部署DNN模型已经很成熟，但在内存受限的边缘设备（如智能手机、智能相机、物联网设备）上部署DNN面临很大挑战，主要问题是内存限制。

云端DNN模型通常基于强大的GPU集群，而**边缘设备的内存非常有限。**例如，一些高端智能手机的内存仅为**12GB**，而GPU服务器可能有80GB的内存。操作系统和应用程序可能进一步将DNN推理的内存使用量限制在**512MB甚至50MB。**据报道，Microsoft Editor中的语法检查器模型必须使用小于50 MB的内存。

移动/边缘设备上内存容量的增长明显滞后于计算能力的进步，而DNN模型的计算和内存需求通常随着模型能力的增加而线性增加。

因此，如何减少DNN推理的内存占用，是决定是否能在这些设备上使用模型的关键。

主流的端设备DL部署方案是**模型定制。**

包括**模型压缩**、**架构设计与搜索**等方法，这些方法可以减小模型的大小，但可能会导致精度下降，并且需要大量的开发工作。

另一种思路是通过**系统设计**来减少内存占用，但现有的DL框架（如MNN、NCNN等）并**未将内存管理作为重点**，通常采用粗粒度的内存池策略，容易导致内存浪费和碎片化。

挑战

层间内存占用不均衡：不同层的内存占用差异较大，最大内存占用的层会成为瓶颈，使得仅对其他较小的层进行内存交换没有太大意义。

内存管理的开销：为了减少内存占用，通常需要额外的操作（如数据交换），这会带来额外的内存I/O开销，尤其是在边缘设备上，I/O带宽有限。

内存预算的动态变化：边缘设备的内存通常是多个应用共享的，内存容量可能会根据不同的使用场景和任务发生变化，因此需要动态适应内存预算的变化。

贡献

FlexNN框架：这是一个高效且适应性强的DNN推理框架，针对内存受限的设备进行了优化。其关键思想包括：

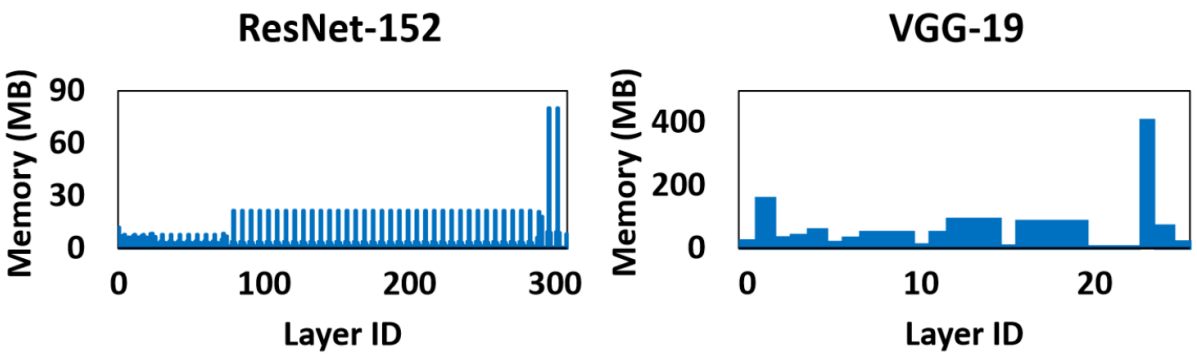
1. 通过联合规划模型执行计划和内存管理计划，优化内存使用。
2. 通过离线预处理减少运行时内存管理开销。

瓶颈感知层切分：对于占用最大内存的层，采用切分技术（slicing），将其拆分成更小的部分，避免单个大层占用过多内存。

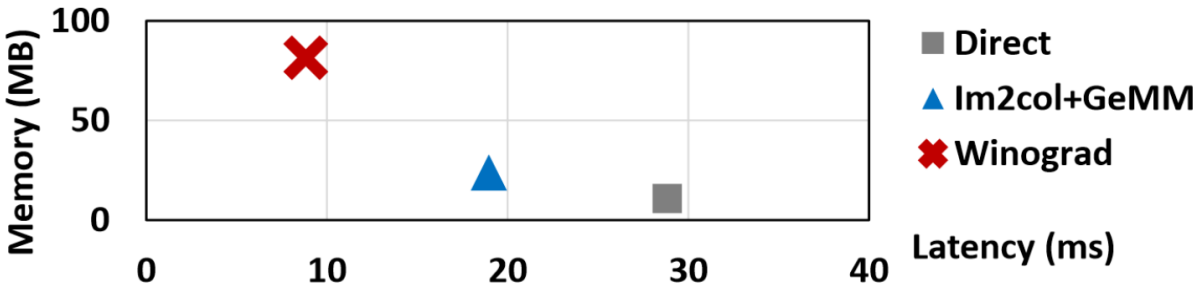
预加载感知内存规划：通过提前规划内存布局，减少内存碎片和I/O等待时间，从而提升效率。

动态内存预算适应：通过离线生成的配置文件和权重，当内存预算发生变化时，系统可以迅速适应，避免大规模的实时计算。

Motivation



DNN模型表现出高度不平衡的逐层内存分布，少数层具有极大的内存占用。



Direct Conv：直接计算卷积结果，计算复杂度较高。

Im2col+GeMM：将输入特征展开，转换为矩阵乘法，更适合硬件加速。

Winograd：通过在预处理阶段转换权重，降低3×3卷积的计算复杂度，计算延迟最低。

在3 × 3 Conv的3个内核之间进行延迟-内存折中，延迟和内存使用率之间存在负相关关系。

层的内存占用主要由三部分组成：**激活值（activations）、权重（weights）和中间数据（intermediates）**。具体来说：

- **激活值**是层的输入和输出。
- **权重**是模型的参数。
- **中间数据**是在计算层输出时产生的临时结果。

内存瓶颈通常出现在**权重**和**展平输入**（如在Im2col+GeMM卷积中）上：

- 在**全连接层（FC）和Winograd卷积层**中，权重通常占据大部分内存。例如，在VGG-19的最大全连接层中，权重占内存的99.97%，而在ResNet-152的最大卷积层中，Winograd卷积的权重占83.63%。
- 在**Im2col+GeMM卷积层**中，展平输入占用了大部分内存。例如，在VGG-19的第二个卷积层中，展平输入占内存的74.75%。

机遇

为了处理层间内存分布不平衡，减小瓶颈层的内存占用，可以进行层内内存划分。

由于不同层和卷积核的内存需求不同，可以使用不同的划分方法。

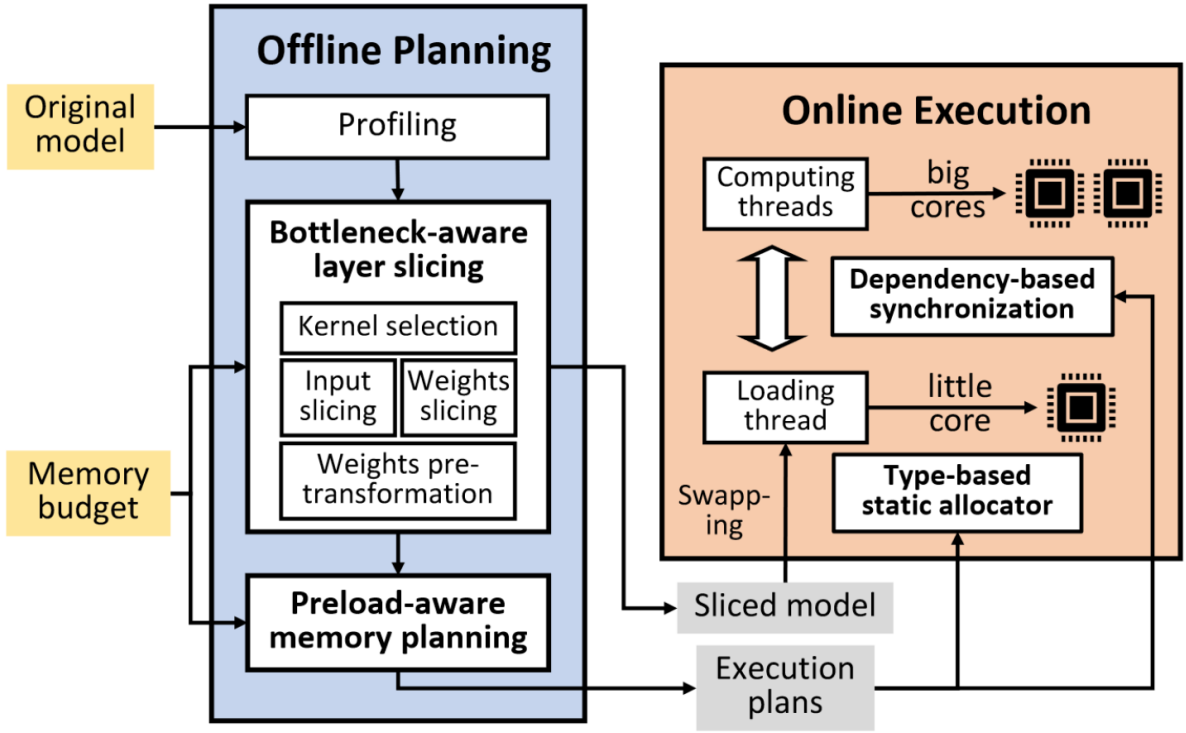
由于不同的卷积核具有不同的延迟-内存权衡，在给定的内存预算下，选择最小化延迟的核是很重要的。

此外，DNN推理负载的独特特性为内存交换提供了更好的设计空间。

1. 与传统软件相比，DNN推理的控制流是确定的，这使得通过提前规划来减少执行时间管理开销成为可能。
2. DNN推理过程中张量的大小和生命周期呈现一定的规律，这些规律为优化内存布局提供了机会

FlexNN设计

总体架构



离线规划阶段：

- **目标**：根据内存预算和给定的模型，进行**切片-加载-计算联合规划。**
- 主要模块：

- **瓶颈感知的层切片（Bottleneck-aware layer slicing）**：通过分析张量的内存大小和生命周期，进行层切片，减少层级内存占用。层切片包括两种切片方式：**权重切片**和**输入切片**，此外还包括**内核选择**和**权重预转换**（某些内核可能需要特定形式的数据（如矩阵形式或扁平化的数组），而在预转换阶段，这些格式转换操作会提前完成，避免了在推理时再进行转换，减少了计算过程中的额外负担），以减少运行时开销。
- **预加载感知的内存规划（Preload-aware memory planning）**：在层切片之后进行，提供详细的**内存规划**。它使用轻量级算法来减少内存碎片和I/O等待时间，并避免大的适应开销。如果内存预算足够大，FlexNN还会将一部分模型权重固定在内存中，进一步减少I/O开销。

在线执行阶段：

- **目标**：根据离线生成的规划和预转换的权重，进行**高效的模型推理**。
- 执行方式：
  - 使用基于依赖的同步机制和基于类型的内存分配器，确保运行时并行执行的正确性。
  - 在初始化时需要进行完整的规划，但在后续的适应过程中通常跳过层切片部分，以避免额外的开销。
- 内存预算变化时的适应过程：
  - 如果内存预算变化，FlexNN首先检查先前的切片模型是否满足新的内存预算。如果满足，直接跳过层切片，只进行内存规划。
  - 如果不满足，仍然需要进行层切片，以确保符合新的内存预算。

瓶颈感知的层切片

目标：通过细粒度的划分，在考虑运行时延迟的情况下，减小每层的内存开销。

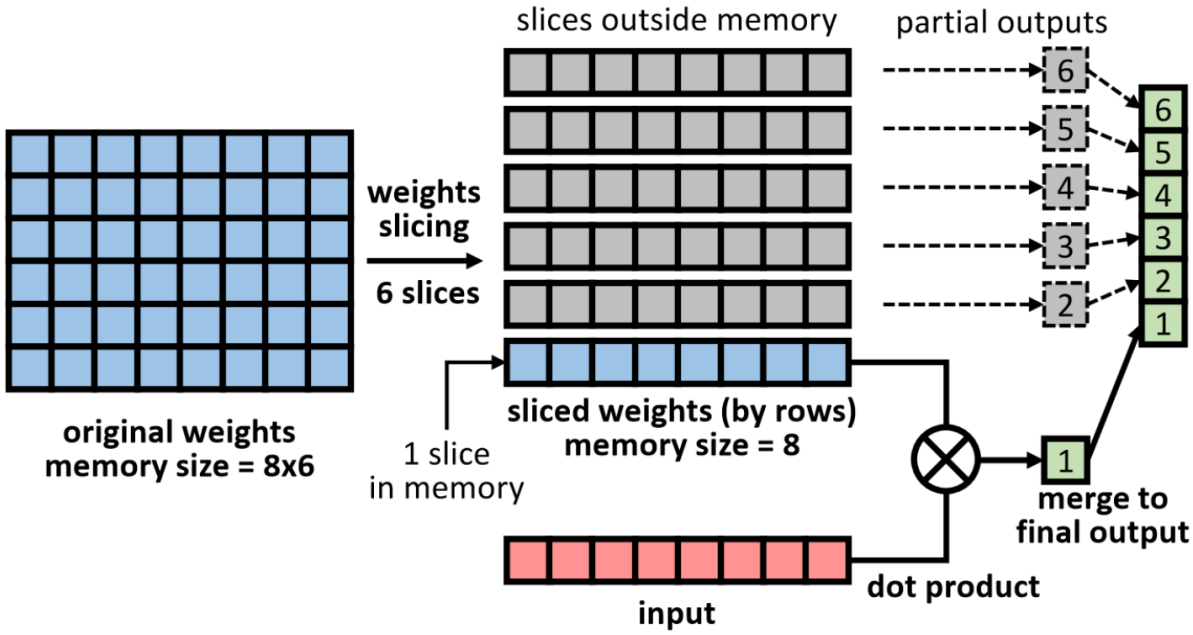
步骤：

1. 核函数选择。
2. 权重/输入切分。基于选择的核进行切分。
3. 权重预转换。避免了运行时处理开销。

权重切分

把层权重划分成切片，并一次向内存加载一个切片来减小内存占用。

切片逐片加载，并与输入相乘，每层产生一个部分输出。将部分输出合并在一起，得到最终输出。



8×6的全连接层（FC）被划分为6个切片，以节省83%的内存使用。

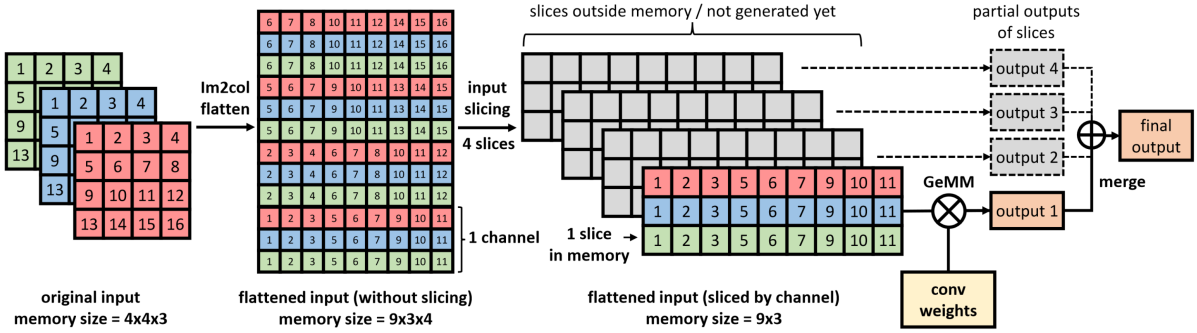
FC的权重大小输入和输出的大小而变化，可以得到非常大的值。对FC的权重进行切片是很有必要的。

权重切片数量应该在内存预算下**最小化**。因为过大的切片数量会显著增加I/O（从磁盘加载切片）和计算（推理）任务切换的调度开销。

输入切分

把扁平化的输入切分，每次只在内存中生成一部分来减少内存占用。

输入扁平化后切分，并与权重逐层相乘以产生部分输出，最后将其合并以获得最终输出



输入切分的切片数应在一个依赖平台的阈值内**最大化**，这与权重切分不同。

因为输入切分中的每个切片都是每次生成的，而不是从磁盘中交换出来，避免了I/O开销。

除非切片的尺寸过小，无法充分利用硬件加速(例如, SIMD)，否则总延迟不会随着切片数量的增加而显著增加。

权重切片更适合于权重占优的层，如FC和Winograd Conv，而输入切片更适合于中间值占优的层，如Im2col + GeMM Conv。

核函数选择

在实际切分之前进行，因为切分方式的选择依赖选择的核函数。

FlexNN的策略：

- **先尝试延迟最优的 kernel**：比如选择 **Winograd Conv**（它通常能够减少延迟）。此时，如果 **Winograd Conv** 在切分后仍然满足内存要求，就使用它。
- **如果延迟最优的 kernel 不满足内存要求**：FlexNN 会切换到一个内存效率更高的 kernel，例如 **Im2col+GeMM Conv**。这种 kernel 在延迟和内存之间做了不同的平衡，通常会用输入切分来减少内存占用。
- **如果 Im2col+GeMM Conv 也无法满足内存要求**，则回退到最基础的 **Direct Conv** 方法。Direct Conv 不做额外的优化，通常内存占用较低，但延迟较高。

权重预转换

在核函数选择之前进行，用于在执行之前转换特定的权重。

这种转换不仅包括对Winograd Conv等内核进行整形或重新格式化，还包括对Winograd Conv等内核进行矩阵乘法运算，这会带来不可忽略的开销。

使得FlexNN运行时能直接加载转换后的权重。

预加载感知的内存规划

动态内存管理策略由于缺乏全局信息，会导致碎片的增加。

静态内存管理策略没有考虑预加载，会导致次优解。

FlexNN使用了预加载感知的静态内存管理策略。

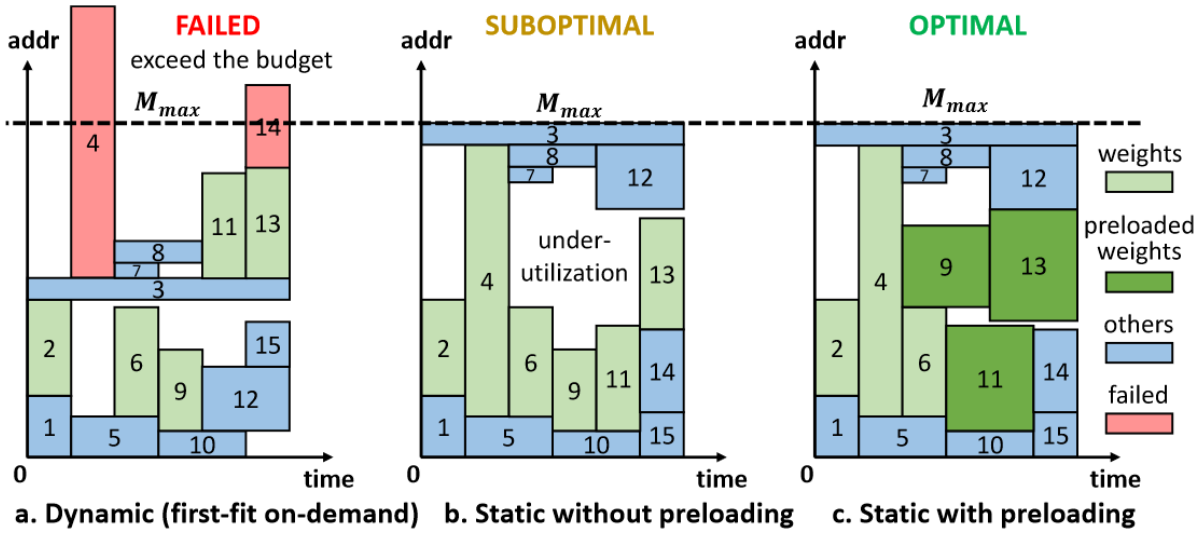
形式化问题

通常来说，内存布局规划问题通常归结为二维装箱(2D Bin Packing，2DBP)问题。

2DBP问题的常见应用：

- 传统的内存布局规划问题通常被形式化为 **2D Bin Packing (2DBP)** 问题。在这种形式化中，**每个张量**（tensor）被抽象为一个矩形（称为“张量块”），该矩形在二维平面上由两个坐标轴表示：**y轴**表示内存地址，**x轴**表示时间（生命周期，即从分配到释放的时间段）。
- 目标是为给定的张量找到一个优化的布局，使得它们在已知生命周期和内存大小的情况下占据最合适的内存地址。

在 **预加载感知的内存规划问题** 中，除了确定张量的内存地址外，还需要决定 **张量的分配时间**（即预加载开始的时间）。



不同内存管理策略的比较。块中的数字表示分配的顺序。

图c中可以看到，权重9，11和13在计算需要之前就开始预加载，因此比图b具有更长的生命周期。

给定以下信息：

- **张量属性**：包括每个张量的分配时间（实际分配时间，没有预加载）s_i、释放时间e_i、内存大小m_i以及内存类型$type(i)$，其中内存类型包括权重（weights）、激活值（activations）和中间数据（intermediates）。
- **内存预算**M_{max}：为所有张量分配的内存总量有一个限制。

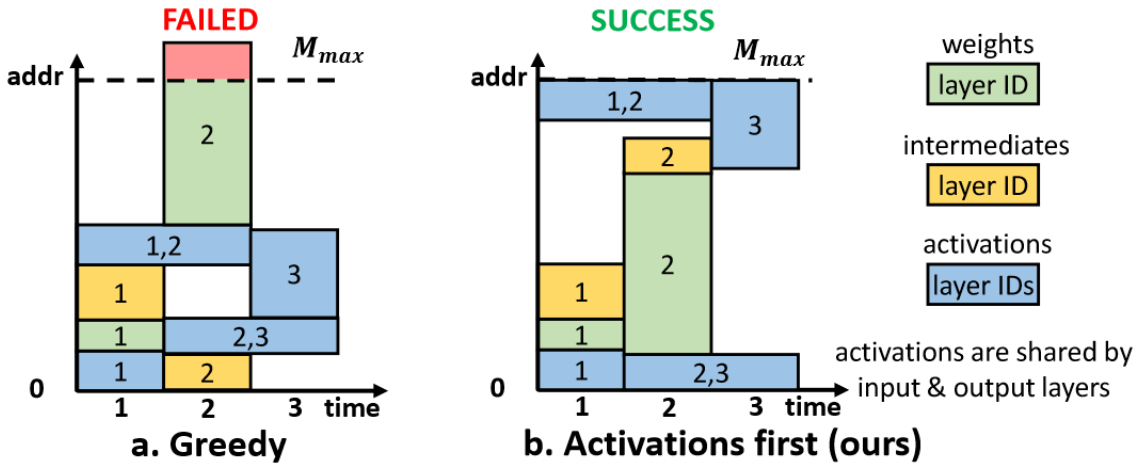
目标是通过找到一个最优的**内存规划Ω** = {⟨$timei, addri$⟩}($timei$是张量i的分配时间，带预加载，$addri$是分配的内存地址)，**最小化推理延迟**，同时满足以下约束条件：

1. **生命周期约束**：对于所有**权重张量**（weights），1 ≤ $timei$ ≤ si，允许预加载。对于**其他类型的张量**，其$timei$ = si。
2. **内存使用约束**：在所有时间步骤内，所有张量的内存总使用量不能超过内存预算（M_{max}）。
3. **内存地址约束**：每个张量分配的内存地址必须在合法的范围内（0 ≤ $addri$ ≤ $Mmax - mi$），且不与其他张量的内存重叠。
4. **内存非重叠约束**：张量的生命周期（分配到释放时间）必须不重叠，或者其内存地址必须不重叠。

规划算法

该算法将统一的缓冲区内内存规划分解为多个步骤：

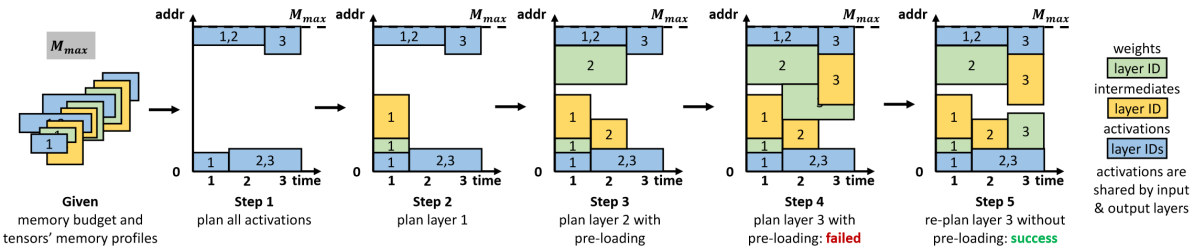
1. 计划激活值：由于激活具备较长的生命周期（因为它们通常作为一层的输出和一层或多层的输入），FlexNN优先规划激活。



先计划激活，避免出现碎片。张量块中的数字表示它们所属的层ID

如图中激活块，在激活优先的放置方法中，它们被放到内存缓冲区的两端，从而在内存中间维持了一个连续的内存块，有效避免了碎片。

2. 逐层计划预加载的权重。FlexNN会先于中间值计划权重。贪心地搜索可用的内存，把权重预加载到内存，最小化权重的分配时间，以减小I/O时间。
3. 逐层计划中间值。对于每层，计划好预加载权重的布局后，贪心地用中间值填满剩余内存。
4. **不预加载下**，逐层重新计划权重和中间值。贪婪的权重预加载策略也可能造成碎片，导致中间值的规划失败，导致预加载感知规划失败。FlexNN从而需要重新计划**没有预加载的**层的权重和中间值的内存放置。



预加载感知内存规划的完整的工作流，以3层为例子，数字代表所属的层ID

在线执行设计

在线执行阶段在正确遵循离线规划阶段确定的计划下，进行实际的模型推理。需要解决规划结果与实际执行之间的两个鸿沟：

1. **逐张量规划和逐层执行之间的鸿沟。**

   在内存规划阶段，内存分配是针对张量级别进行精细化规划的，这涉及张量之间的依赖关系。然而，实际的推理执行是层级级别的，涉及多个张量，且这些张量可能有不同的生命周期。如果在执行层时没有精确控制每层的加载和计算顺序，**可能会导致张量级别的内存冲突。**例如，某些张量的内存可能被提前加载并计算，而其他张量可能还没有准备好。

2. **逻辑时间与实际执行时间之间的鸿沟**：

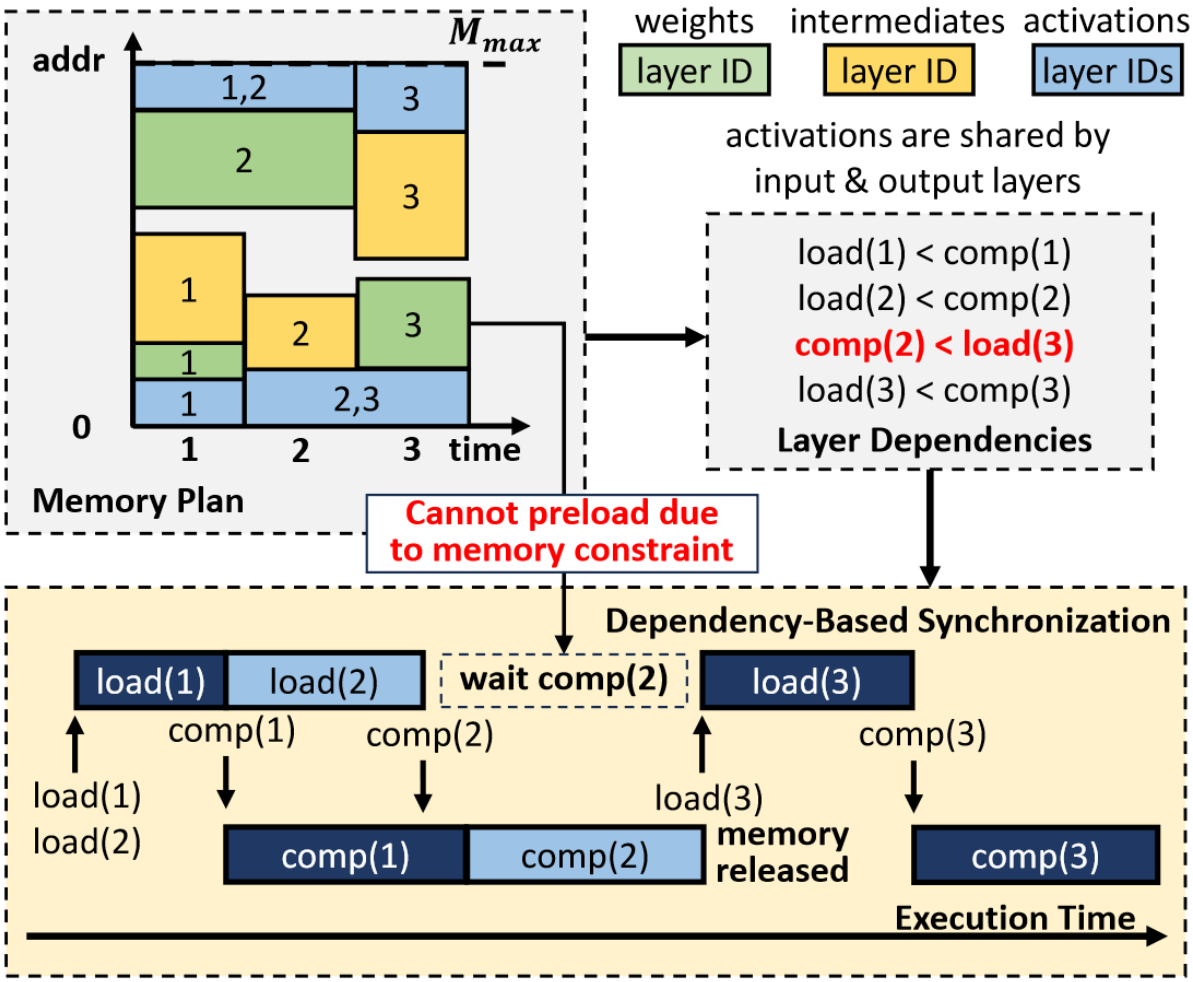
   在内存规划阶段，计划中仅确定了每个张量的分配和释放的逻辑时间（即计划中的时间点）。但实际执行时，会受到系统调度、线程执行顺序等因素的影响，操作的时间可能会有所不同，这种不确定性可能会改变张量的实际分配顺序，导致错误的内存分配结果。

基于依赖关系的同步方案：

通过引入基于张量之间依赖关系的同步方案来调度层的加载和计算。这意味着系统会确保张量在正确的时间点加载，并且按照计划的顺序进行计算，从而避免因顺序错误而引起的内存冲突。

核心思想：

1. **分离加载和计算任务**：
    - 在每一层的执行过程中，FlexNN 将其分为加载任务和计算任务：
        - **加载任务** 负责加载权重。
        - **计算任务** 负责执行其他计算操作。
    - 通过 **任务队列** 管理这些任务：加载任务队列和计算任务队列，计算线程和加载线程分别从各自的任务队列中贪婪地获取任务，并确保执行顺序符合层之间的依赖关系。
2. **任务级依赖关系**：
    - 依赖关系（Dependency）：被用来定义加载和计算任务之间的执行顺序。具体来说，有两种类型的依赖关系：
        1. **加载任务在计算任务之前执行**：用符号 `load(i) < comp(i)` 表示，意味着第 i 层的计算依赖于第 i 层的加载。
        2. **计算任务在加载任务之前执行**：用符号 `comp(i) < load(j)` 表示，意味着如果第 i 层的计算与第 j 层的加载操作有内存地址交集，那么第 j 层的加载依赖于第 i 层的计算。
3. **依赖关系的实现**：
    - **依赖关系1**（加载在计算之前）是自然成立的，对于所有层，都会遵守。
    - **依赖关系2**（计算在加载之前）可以通过简单的后处理从内存规划结果中获得，这样可以在执行过程中根据内存分配的情况决定任务的顺序。



3层模型的基于依赖的同步实例。第3层的加载依赖于第2层的计算，因为它们有内存地址的交集

若预加载3的权重块，会导致3的中间值块无法安放，出现内存冲突，所以内存计划中的**3的权重块是没有预加载的**，所以它的加载时间一定是在2的计算完成之后的。所以comp(2)<load(3)

若没有依赖关系的同步机制，第三层load时，会出现张量级冲突。

基于类型的静态分配器：

使用基于类型的静态分配器来确保内存分配的正确性。这一分配器确保在实际执行时，张量的内存分配正确，遵循计划中的内存地址和生命周期，特别是在多线程执行时。核心思想是**确定每种类型张量的分配顺序。**

分配线程的区分：

- 在 FlexNN 中，每种类型的张量由不同的线程分配内存：
  - **权重张量（Weights）**：内存只在 **加载线程（loading thread）** 中分配。
  - **激活张量（Activations）** 和 **中间结果张量（Intermediates）**：内存只在 **计算线程（computing thread）** 中分配。
- 这样一来，每种类型的张量内存分配顺序就被固化下来，不会出现混乱。

张量类型和分配顺序的固定性：

- 每种张量的内存分配顺序是固定的，因为每个线程只负责自己类型张量的内存分配。
- 为了确保内存分配的正确顺序，**FlexNN通过张量的类型和在该类型中的计数来唯一标识每个张量**，例如 **Weights-5** 就表示 **权重张量中的第 5 个** 张量。

内存分配地址的映射：

- 张量的分配顺序通过 **轻量级的后处理**（post-processing）内存规划结果来获得。通过这种方式，可以快速确定张量 ID 和分配地址之间的映射关系。

实现

FlexNN原型实现：

- FlexNN是基于NCNN（一个高效的推理引擎）实现的原型，代码增加了12.3K行。
- NCNN是一个开源推理引擎，特别针对Arm CPU进行了优化，广泛支持各种实际的移动应用程序。

选择NCNN的原因：

- **优化性能**：NCNN针对移动设备进行了高度优化，性能超过了许多其他框架，如TFLite。
- **支持的模型多**：支持多种深度学习模型，如CNN、RNN、LSTM、Transformer等。
- **社区支持和代码结构清晰**：NCNN拥有较好的社区支持和清晰的代码结构，方便开发。

FlexNN技术的通用性：

- 虽然实现是基于NCNN，但FlexNN的技术关注于DNN推理过程中的数据加载和计算，因此可以很容易地移植到其他框架中。

框架修改：

- **生命周期感知的内存规划器**：这个模块是框架独立的，不需要修改框架。
- **离线规划**：使用NCNN的模型写入工具进行层切分，修改计算图，并写入转化后的权重。
- **运行时引擎**：实现了基于依赖关系的同步机制的并行预加载流水线，并将原NCNN内存分配器修改为基于类型的静态内存分配器，改变了原有的内存管理方案。静态分配器在初始化时分配一个连续的缓冲区，并在推理过程中管理该缓冲区。
- **修改NCNN操作符**：例如，通过调整Im2col+GeMM卷积内核实现输入切分。

支持Armv8-A和NEON架构：

- **Armv8-A架构**：FlexNN实现目标是Armv8-A CPU，这是目前智能手机中广泛使用的标准架构。尽管如此，FlexNN的设计是可以兼容其他架构的。
- **NEON SIMD扩展**：NEON是Armv8-A的高级SIMD架构扩展，支持每条指令最多进行4次32位操作，NCNN的内核使用8 × 4和8 × 8等形状的基本计算块，并采用NEON加速。因此在**切分层时**，会尽量保持切分后的输入/权重的宽度/高度/通道数**为8的倍数**，以充分利用NEON加速，避免性能下降。

实验评估

实验设置

评估模型

- FlexNN在六个广泛使用的深度神经网络（DNN）模型上进行评估，包括：
  - **ResNet-152**
  - **VGG-19**

    - **Vision Transformer (ViT)**
    - **GPT-2**
    - **MobileNetV2**
    - **SqueezeNet**
- 评估中使用了这些模型的预训练权重（FP32）和随机输入，因为FlexNN不会修改模型的权重。

平台

- 实验在两种类型的边缘设备上进行评估：**单板计算机**和**智能手机**，这些设备具有不同的硬件规格，所有设备都配备了Armv8-A架构的CPU。
- **Pixel 6 Pro**：使用两个大核心进行计算，使用一个中等核心进行数据加载。
- **Mix 2S**：使用四个大核心进行计算，使用一个小核心进行数据加载。
- **Raspberry Pi**：没有小核心，所有计算和加载任务都由大核心处理。

Device Name	CPUs	RAM (GB)
Google Pixel 6 Pro	2x2.80 GHz Cortex-X1 2x2.25 GHz Cortex-A76 4x1.80 GHz Cortex-A55	12
Xiaomi Mi Mix 2S	4x2.8 GHz Kryo 385 Gold 4x1.8 GHz Kryo 385 Silver	6
Raspberry Pi 4B	4x1.8 GHz Cortex-A72	8

设备规格

Metrics

- 内存使用：通过Linux pmap命令测量。
- 推理延迟：
- 能量消耗：将电池的电流和电压相乘，可以通过Linux下的sysfs接口获取。

Baselines

所有的baselines都是基于NCNN的，因为：

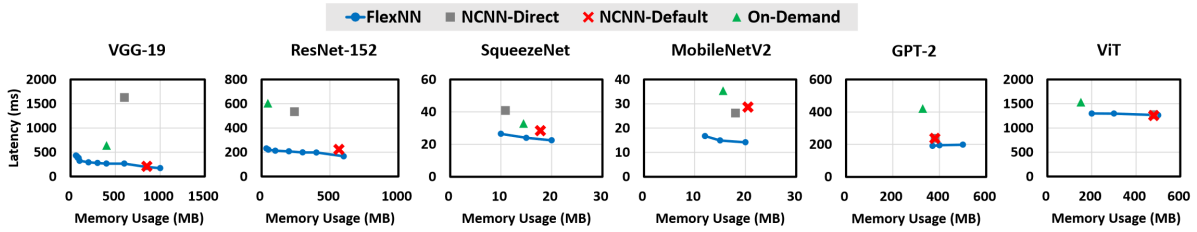
- FlexNN是基于NCNN实现的，并且NCNN在移动端框架中表现优异。
- FlexNN是第一个将内存优化作为优先考虑的移动推理框架，因此没有类似的工作可以直接比较。

- **NCNN-Default**：使用默认配置的NCNN，作为一个基准方法。
- **NCNN-Direct**：通过禁用Im2col+GeMM和Winograd内核来减少内存使用，这两种内核通常会增加内存消耗以降低延迟。
- **On-Demand**：在NCNN上实现的层级交换和按需加载策略，以减少内存占用。

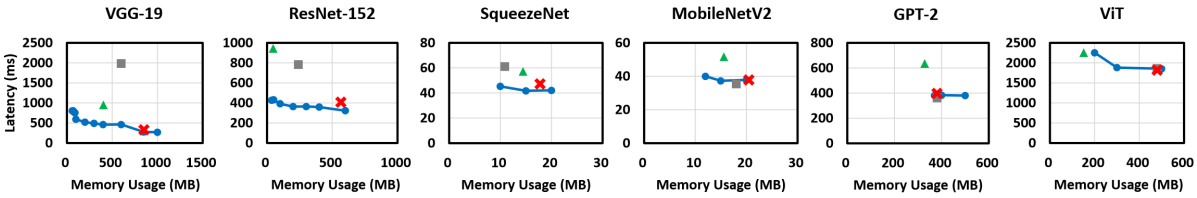
所有基准方法与FlexNN一样，使用相同数量的大核心（big cores）进行计算。

baselines不使用小核心（little cores），因为如果同时使用大核心和小核心进行计算，负载不均衡会导致推理速度变慢（例如，在Mix 2S上慢48%）。

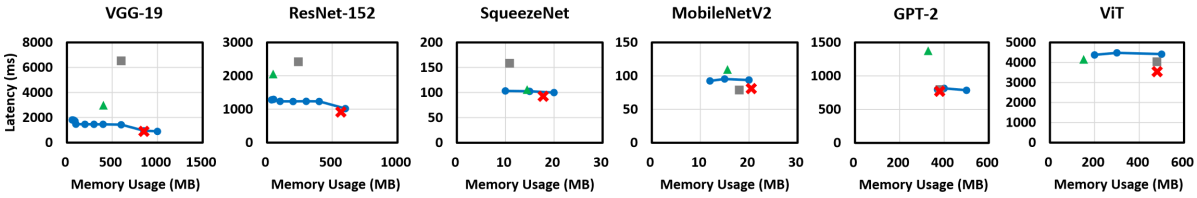
实验结果



(a) Google Pixel 6 Pro



(b) Xiaomi Mix 2S



(c) Raspberry Pi 4B

不同的设备和模型上的端到端的延迟和内存结果

FlexNN是一条曲线，而baselines的结果是一个点，因为baselines的内存预算固定。

峰值内存的减少

FlexNN支持比NCNN-Default和NCNN-Direct更低的内存预算，在大多数模型上也低于On-Demand策略。

On-Demand的内存节省：作为一种简单的流式处理方法，On-Demand策略在大多数模型上比其他基准方法使用更少的内存。

FlexNN的层切分技术：FlexNN通过层切分技术进一步减少了每层的内存占用，相比On-Demand策略，进一步降低了整体的内存使用。

延迟的减少

在相同的内存预算下，FlexNN始终如一地获得比基线更低或至少相当的延迟。

在低内存预算下，FlexNN通过权重预转换以及预加载，维持了和NCNN-Default相当的延迟。在ResNet - 152，Pixel 6 Pro上仅以3.64 %的延迟增加为代价，实现了93.81 %的峰值内存减少。

当内存预算足够大时，FlexNN的延迟甚至优于NCNN - Default，主要因为当给定与NCNN - Default相同的内存预算时，FlexNN可以在内存中固定的所有权重，从而完全避免了I/O开销。

不同模型和设备的变化

模型类型的差异：

- **CNN模型**：FlexNN在卷积神经网络（CNN）模型中表现良好，能够显著提高性能，尤其是在不同规模的CNN模型上。
- **Transformer模型**：在基于Transformer的模型中，FlexNN的改进效果并不显著。这表明，Transformer模型的性能提升受到某些因素的限制。

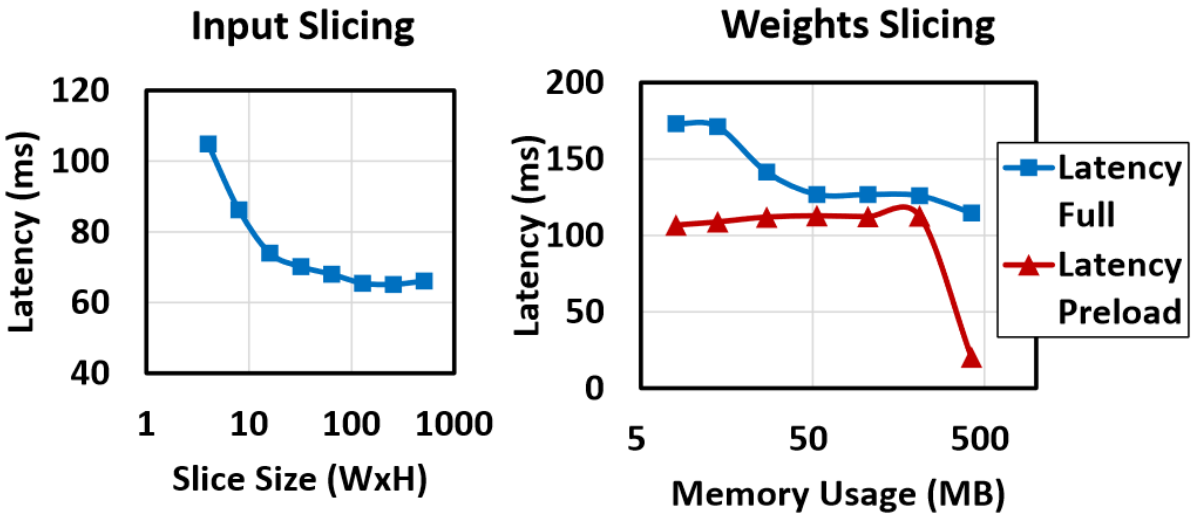
设备差异的影响：

- **Raspberry Pi表现较差**：FlexNN在Raspberry Pi上的性能稍逊于其他设备。这可能与Raspberry Pi的硬件性能瓶颈有关。

瓶颈变化的原因：

- **计算瓶颈与I/O瓶颈**：在大多数情况下，设备的DNN推理主要受限于计算（计算瓶颈），此时FlexNN能够通过计算时间来隐藏加载时间，从而提高性能。
- **I/O瓶颈**：然而，对于以I/O为主的模型（如Transformer模型），或者在设备存在I/O性能瓶颈时（例如，Raspberry Pi的平均读取速度为257 MB/s，而Mix 2S为718 MB/s），推理任务会变成I/O受限。这时，加载时间无法被计算时间隐藏，导致延迟无法进一步减少。

延迟-内存的权衡



在Pixel 6 Pro上测量了单层切片的延迟和内存权衡。权重切片在FC上进行测试，输入切片在3 × 3 Conv上进行测试。

- 对于输入切片，随着切片尺寸的增大，延迟的减小幅度变小。本文采取32的大小作为实现，为预加载预留更多的内存空间。
- 对于权重切片，当内存预算减小时，没有预加载策略的方案的延迟显著增加，而有预加载策略方案的延迟几乎不变。

系统开销

离线计划开销

Model	Memory Budget (MB)	Transformed Weights Storage (MB)	Layer Slicing Cost (ms)	Profiling Cost (ms)	Memory Planning Cost (ms)
VGG-19	100	781	2,458.96	689.25	5.27
ResNet-152	100	547	2,228.76	540.94	864.67
Vision Transformer	300	337	888.84	1,438.84	390.55

FlexNN的存储和时间开销

离线规划的组成：

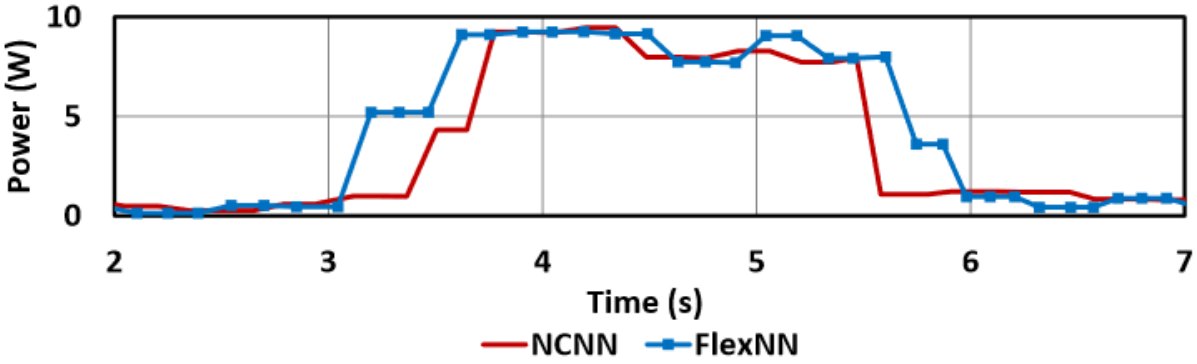
- **存储开销**：包括存储转换后的权重（transformed weights），这些通常需要足够的存储空间，通常为几十GB到几百GB，现代边缘设备通常能够满足这一要求。
- 时间开销：主要包括以下几个部分：
  - **profiling和layer slicing**：这些操作通常只在初始化时执行，因此它们的时间开销仅发生一次。
  - **内存规划（memory planning）**：需要在每次内存预算发生变化时重新执行，所以这是需要重复执行的部分。

重新切片和重新规划的额外开销：

- 只有当切片结果不符合新的内存预算，或出现内存分配错误时，才需要重新进行切片和内存规划。
- 由于方法能够有效地控制内存使用，因此这种情况发生的频率应该非常低。

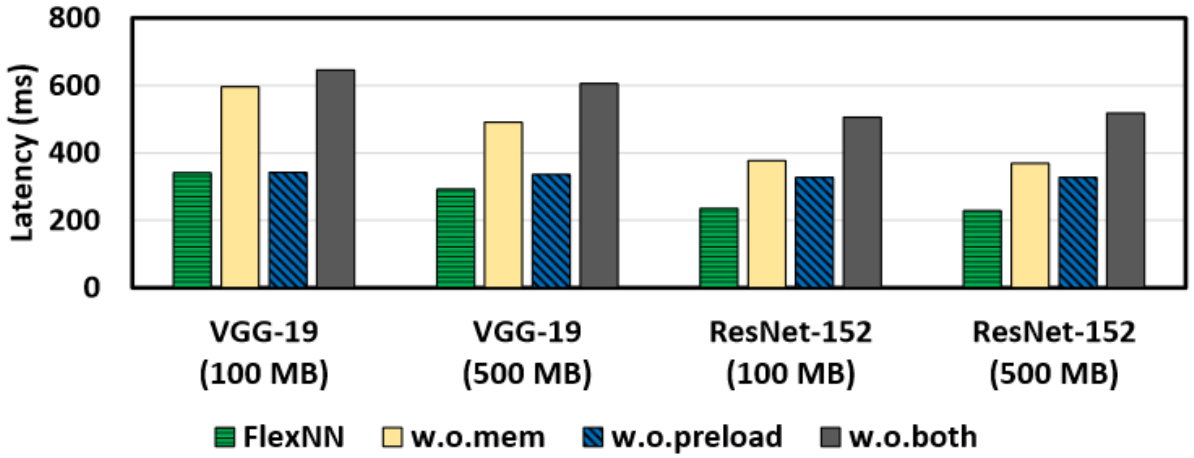
运行时I/O开销

前面已经做了延迟的端到端结果对比，这里主要关注**能量开销。**



FlexNN的峰值功耗与NCNN相似，但FlexNN较大的推理延迟导致总能耗较高

消融实验



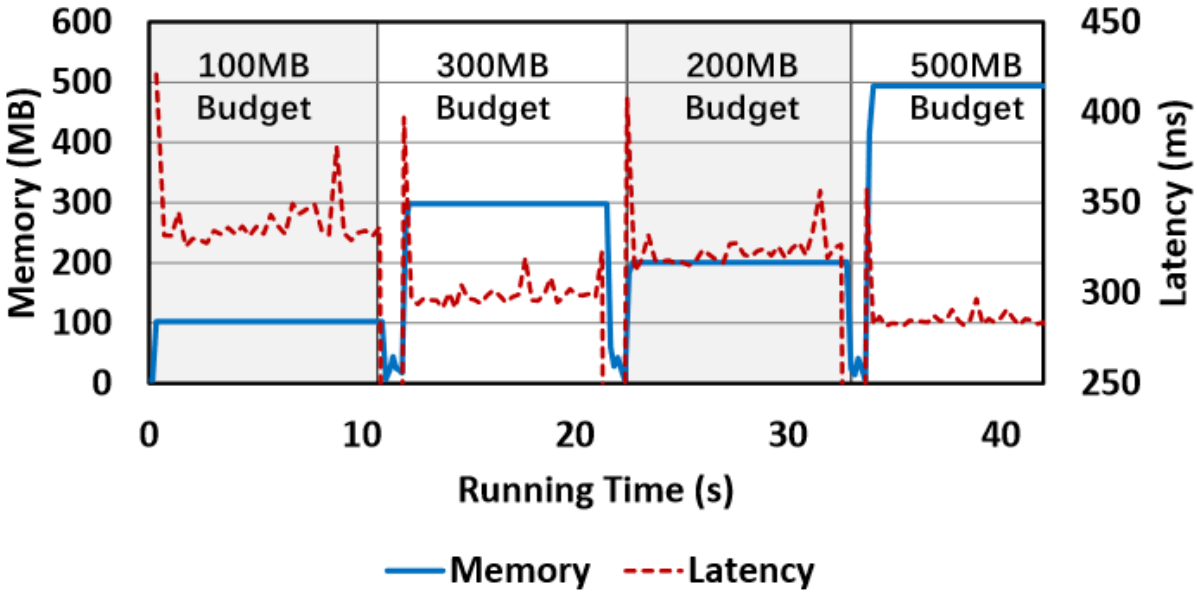
' w.o.mem '表示没有静态内存管理的FlexNN；

" w.o.preload "表示无并行预载的FlexNN；

' w.o . both '代表没有这两个组件的FlexNN。

在给定的内存预算下，静态内存管理和并行预加载都有效地降低了推理延迟

对内存预算改变的适应



FlexNN对每个内存预算在VGG - 19上进行32次推理。

当内存预算改变时，FlexNN暂停推理过程，释放内存，为新的内存预算生成执行计划，重新分配内存，最后恢复推理。整个适应过程总共耗时约1秒。

由于没有预热，每次适应后的第一个循环由于冷启动开销而具有较长的延迟。

讨论

不同的内存瓶颈:

- **FlexNN** 在处理层级内存瓶颈时表现良好，尤其是权重或输入作为瓶颈时。然而，当 **激活值** 成为主要内存瓶颈时，效果可能有限。这是因为当前设计不支持激活值切片（activation slicing），且输入切片仅限于类似 **Im2col+GeMM** 的展平输入。为了解决这一问题，未来可以通过支持激活分区和交换（activation partitioning and swapping）来改进。

不同的模型：

- FlexNN目前主要针对典型的 CNN 模型（如 VGG、ResNet、MobileNet）进行优化，因为这些模型在移动设备上应用广泛。虽然也在 Transformer类型的模型（如 ViT、GPT-2）上进行了测试，但效果不显著。原因有两点：
    1. **Transformer** 模型中的多头注意力（MHA）层的切片支持不如卷积层，需要更多的工程工作来改进。
    2. **Transformer** 模型的推理过程中需要更多的权重加载，因此在基于流式推理的情况下，会带来更大的 I/O 开销，使得内存交换机制的应用变得更加困难。

不同的精度：

- 尽管当前实现只支持基于 **FP32** 的模型，**FlexNN** 的设计与数据精度无关，能够支持其他精度（如 **FP16** 或压缩模型）。但是，操作符的移植仍然需要一定的工程工作。

不同的硬件：

- FlexNN目前主要针对移动 CPU进行优化，因为移动设备和嵌入式设备的 AI 应用大多依赖 CPU。然而，该设计是通用的，能够迁移到不同的硬件平台（如移动GPU、NPU、DSP和 Tensor Cores），但迁移时需要考虑以下因素：
    1. 需要低级 API 支持以实现细粒度的内存管理。
    2. 利用异构硬件会增加数据迁移的成本，流式推理设计带来的额外 I/O 操作也需要考虑。
    3. **FlexNN** 可能会增加能耗，这对于电池供电的设备是一个限制。

与其他内存优化技术的对比：

- 提到一些与 **FlexNN** 相似的技术，如 **Melon**，它通过离线规划和生命周期感知的内存池来减少设备训练的内存占用。但这些技术缺乏联合规划（layer partitioning、内存布局、计算和加载重叠），直接应用这些方法可能导致次优性能。尽管如此，它们为边缘设备的内存优化提供了有价值的见解。

模型定制化和内存受限推理：

- 为了支持资源受限的设备上进行 DNN 推理，很多研究致力于模型定制技术，如模型压缩、有效的模型结构设计、神经架构搜索（NAS）等。通过剪枝（pruning）和量化（quantization）等方法来减少模型参数、计算量和内存需求。同时，硬件感知的 **NAS** 可以自动搜索最适合给定平台和任务的模型架构。面对内存预算动态变化的挑战，研究者们还提出了动态缩放模型的方法。
- 尽管这些技术能有效减少计算和内存需求，但可能会影响模型的能力和鲁棒性。与这些技术相比，**FlexNN** 和它们是正交的。通过将定制模型与 **FlexNN** 结合，可以进一步减少内存占用。